Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 61 (2006) 16–26

Science of Computer Programming

www.elsevier.com/locate/scico

Detection of anomalies in software architecture with connectors

Michael E. Shin*, Yan Xu, Fernando Paniagua, Jung Hoon An

Department of Computer Science, Texas Tech University, Lubbock, TX 79409-3104, United States

Received 4 March 2005; received in revised form 14 September 2005; accepted 7 November 2005
Available online 9 March 2006

Abstract

This paper describes an approach to detecting anomalies in a software architectural style that is structured with components and connectors between the components. Each component is designed with tasks (concurrent or active objects), connectors between tasks, and passive objects accessed by tasks. Anomalies in the software architecture are detected twofold by each Component Monitor, which supervises objects in a component, and by a System Monitor, which monitors message communications between components. The monitors encapsulate the specifications of objects being monitored, which are represented using statecharts. The execution of statecharts in the monitors depends on notification messages from connectors between tasks, passive objects accessed by tasks in a component, and connectors between components.

Keywords: Anomaly detection; Software architecture; Connector; Component; Notification message

1. Introduction

The detection of anomalies in dependable systems is crucial to high quality service in those systems. Anomalies in a system can result from different types of causes such as design faults, system failure, malicious attacks, physical damage, low performance, or network congestion [8]. The important factors in the mechanism for anomaly detection are (1) how fast the mechanism detects anomalies and (2) how accurately it locates them in a system. Thus very dependable systems are in need of anomaly detection mechanisms that are capable of providing both high speed detection and accurate location of anomalous objects.

Several approaches to anomaly detection for dependable systems have been suggested in [3,8–10,7,4], which may provide partial solutions from the perspective of quality factors of the mechanisms for anomaly detection—speed and accuracy. Time testing, referred to as heartbeating [3,9,10], can be used to check if a component or system is anomalous, but it may fail to locate where an anomaly is in a component or system. The detection mechanism depending on exceptions [7] may not handle unanticipated, state-dependent anomalies.

This paper describes an approach to detecting anomalies in a software architectural style structured with components and connectors, which can meet quality factors of detection mechanisms such as speed and accuracy of anomaly detection. This begins by describing a software architecture style in Section 2. Section 3 describes the

* Corresponding author. Tel.: +1 806 742 3527.
E-mail addresses: Michael.Shin@ttu.edu (M.E. Shin), yan.xu@ttu.edu (Y. Xu), fernando.paniagua@ttu.edu (F. Paniagua), jh.an@ttu.edu (J.H. An).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.11.002

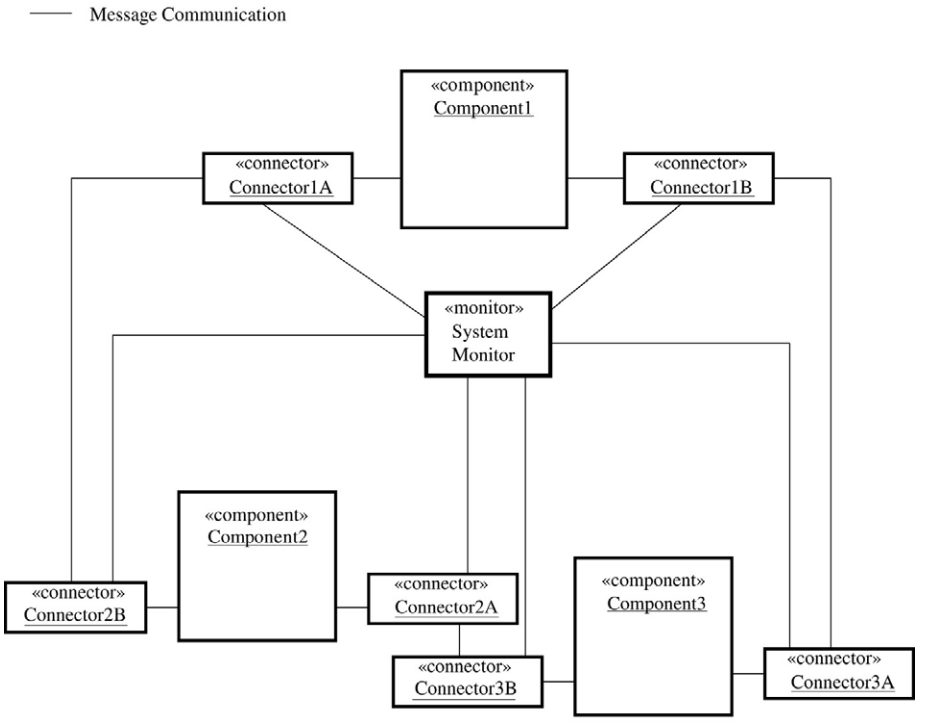

Fig. 1. Overview of software architecture with connectors.

overview of the approach suggested in this paper. Sections 4 and 5, respectively, describe the detection of anomalies within a component and in interactions between components of the software architecture. Section 6 describes the evaluation of our approach. Section 7 describes related work. Finally, Section 8 concludes this paper.

2. Software architecture with connectors

A software architecture [2,12] can be modeled by means of components and their interactions via connectors between the components. A component provides functional services to others. Messages passing between components are delivered via connectors, which synchronize the message communication between components. Fig. 1 depicts an overview of the software architecture with connectors, which is structured with three components, Component1, Component2, Component3, and their connectors. For example, Component1 has Connector1A and Connector1B for message communication with Component2 and Component3.

Each component in the software architecture with connectors can be designed with active objects referred to as tasks, passive objects accessed by tasks (e.g., entity objects storing data), and connectors between tasks [5]. Fig. 2 depicts objects in a component. A task has its own thread of control, initiating actions that affect other active and passive objects [5]. Unlike a task, a passive object has no thread of control; thus it cannot initiate any active objects. But a passive object is invoked by tasks and can invoke other passive objects. Because a passive object does not have its own thread, it performs its operations using the thread of the task that invoked the object. Tasks in a component can communicate with each other through connectors. On behalf of a task, connectors send messages to and receive them from other tasks. Like a connector between components, a connector between tasks within a component encapsulates the synchronization mechanism for message communication.

3. Approach to anomaly detection

An anomaly in the software architecture with connectors can be hidden in components and interactions between the components. The software architecture with connectors is anomalous if components or interactions between the components in the architecture do not behave as specified. The specifications of components and interactions between components in the software architecture are described using statecharts, which model the dynamic aspects of components and interactions between components by means of states and transitions with events.

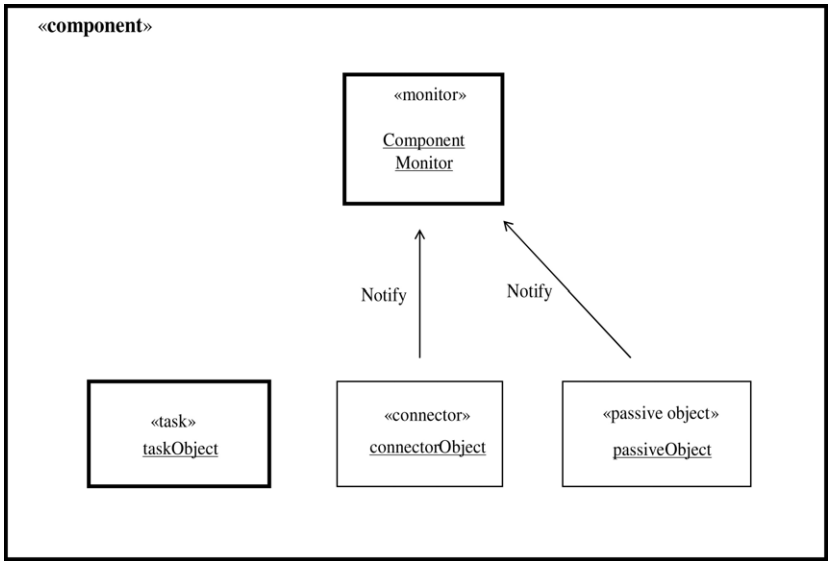

Fig. 2. Objects in a component of software architecture with connectors.

Anomalies in the software architecture with connectors are detected at two levels—at the level of each component and at the level of message communication between components. Anomalies of tasks, passive objects accessed by tasks, and connectors between tasks in a component are detected by a Component Monitor in the component, which traces the behavior of task threads in the component. Fig. 2 depicts a Component Monitor in a component. A task with its own control may request a connector to deliver a message to another task, while it may access a message in a connector to process it. A task may invoke an operation in a passive object to update information. Each task thread, which calls operations in connectors between tasks and/or passive objects accessed by tasks, is monitored using notification messages from the connectors and passive objects. Whenever a connector between tasks and an operation of a passive object are initiated by a task, they notify the component monitor, which uses those notification messages to detect anomalies of tasks, connectors between tasks, and passive objects accessed by tasks. To trace task threads in a component, a component monitor encapsulates statecharts describing the behavior of each task thread. A task, connector between tasks, or passive object accessed by tasks, which is controlled by a task thread, is anomalous if a statechart for the task thread does not execute within a specified time interval.

Anomalies in interactions between components in the software architecture with connectors are detected by a System Monitor using notification messages from connectors between components. Fig. 1 depicts the System Monitor in the software architecture with connectors, which is involved in detecting anomalies in message communications between components. Each message delivered between components is monitored using notification messages from sender and receiver connectors. A sender connector between components notifies the System Monitor of the status of messages sent, while a receiver connector notifies the System Monitor of the status of messages received. These notification messages are used to execute statecharts, which are encapsulated in the System Monitor to trace message communications. Similar to anomaly detection in a component, a sender connector, receiver connector, or interaction between components is anomalous if the statechart associated with a message communication between components does not execute within a specified time interval.

The approach to detecting anomalies intra-component and inter-component suggested in this paper can find out immediately and accurately where an anomaly is in a component or in an interaction between components when statecharts encapsulated in a Component Monitor or System Monitor do not make a transition as expected within a specified time interval. This is because the statecharts describe the behavior of tasks, connectors between tasks, passive objects accessed by tasks, connectors between components, and the network between connectors.

However, the approach in this paper makes some assumptions. Each Component Monitor and the System Monitor need to be robust enough not to become anomalous. Our approach does not include a way of detecting anomalies in the monitors. Our approach also asks for a software architecture pattern described in Section 2, structured with

connectors and components, in which each component should be organized with tasks, connectors between tasks, and passive objects accessed by tasks.

4. Anomaly detection within a component

The threads of tasks in a component are traced to detect anomalous objects on the basis of messages from connectors between tasks, passive objects accessed by tasks, and connectors between components. A connector between tasks in a component not only synchronizes message communication between tasks, but also notifies the Component Monitor when a task calls an operation in the connector to send a message to other tasks or read a message from the connector [13]. The notification message from a connector is used to confirm that the task calling an operation in the connector is working normally up to that point. If the notification message from the connector does not arrive within an expected time interval, the Component Monitor presumes that the task, which was supposed to call an operation in the connector, is anomalous.

A connector between tasks also notifies the Component Monitor of the execution status of its operation called by a task at the end of the execution [13]. This notification message is used to detect the anomalies of the operation in a connector, which is called by a task. If a connector does not notify the execution status of its own operation within a specified time interval, the Component Monitor suspects that the operation called by a task has not been executed successfully.

A passive object accessed by several tasks in a component also needs to notify the Component Monitor both when its operation is invoked by a task and when the operation has been executed successfully [13]. Similar to connectors, the trace of a task thread within a passive object can determine if a task that should invoke an operation in a passive object is anomalous, and if the operation of the passive object invoked by a task is executed successfully.

The Component Monitor can immediately point out an anomalous operation in connectors or passive objects, and an anomalous task, if a statechart in the Component Monitor does not make a transition as specified. The Component Monitor contains the statecharts for threads of all tasks in a component, monitoring the behavior of tasks, connectors, and passive objects accessed by tasks, by tracing state transitions of each task thread. A statechart in the Component Monitor describes the behavior of a task thread, which calls operations in connectors to communicate messages with other tasks, and invokes operations in passive objects to access data in the objects, as well as processes messages read from connectors. The states and transitions in a statechart encapsulated in the Component Monitor can be classified into states and transitions associated with the behavior of each task, connector between tasks, and passive object accessed by tasks. Thus, a monitor in each component can indicate where an anomalous object is in the component when a transition in a statechart associated with an object does not occur within a specified time interval.

The Component Monitor detecting anomalies in Component1 (Fig. 1) is depicted in Fig. 3 using the collaboration model of the Unified Modeling Language (UML) [1,11]. Suppose that Component1 is composed of Component1 Monitor, Task1, Task2, Connector1, Connector2, and Passive Object1. The Component1 Monitor monitors Task1, Task2, Connector1, Connector2, and Passive Object1 in Component1. Fig. 4 depicts the statecharts encapsulated in the Component1 Monitor, which execute concurrently with each other to trace the threads of Task1, Task2, and partially Connector1A. The thread associated with Connector1A is monitored to detect anomalies in an operation in Connector1, which is called by Connector1A. Task1 and Task2 are inner tasks in Component1. The message sequence A1 through A19 describes a service provided by Component1 and the messages notified to the Component1 Monitor.

The message sequence A1 through A3 in Fig. 3 is initiated by the thread of Connector1A. When Connector1 receives a message, “Message1” (A1 in Fig. 3) from Connector1A, it notifies the *Component1 Monitor* of the message arrival (A2 in Fig. 3), which makes a transition in the statechart ((a) in Fig. 4) from the Idle state to the *Placing Message* state. After placing a message in a queue or buffer, the Connector1 again notifies the Component1 Monitor of a message “Message Placed” (A3 in Fig. 3), which is used to determine if the operation in Connector1 called by Connector1A has been executed completely.

The thread of Task1 is monitored by means of the message sequence A4 through A13 in Fig. 3. The Task1 reads a message from the queue or buffer (A4 and A7 in Fig. 3) and Connector1 notifies the fact to the *Component1 Monitor* through the messages *Read Invoked* (A5 in Fig. 3) and *Message Read* (A6 in Fig. 3). These notified messages result in transitions in the statechart for the Task1 thread ((b) in Fig. 4), making *Component1 Monitor* expect the next message “*Update Invoked*” (A9 in Fig. 3) from Passive Object1. The message, *Read Invoked* (A5 in Fig. 3), makes the statechart transition ((b) in Fig. 4) from the *Idle* state to the *Reading Message* state. This transition in the statechart means that

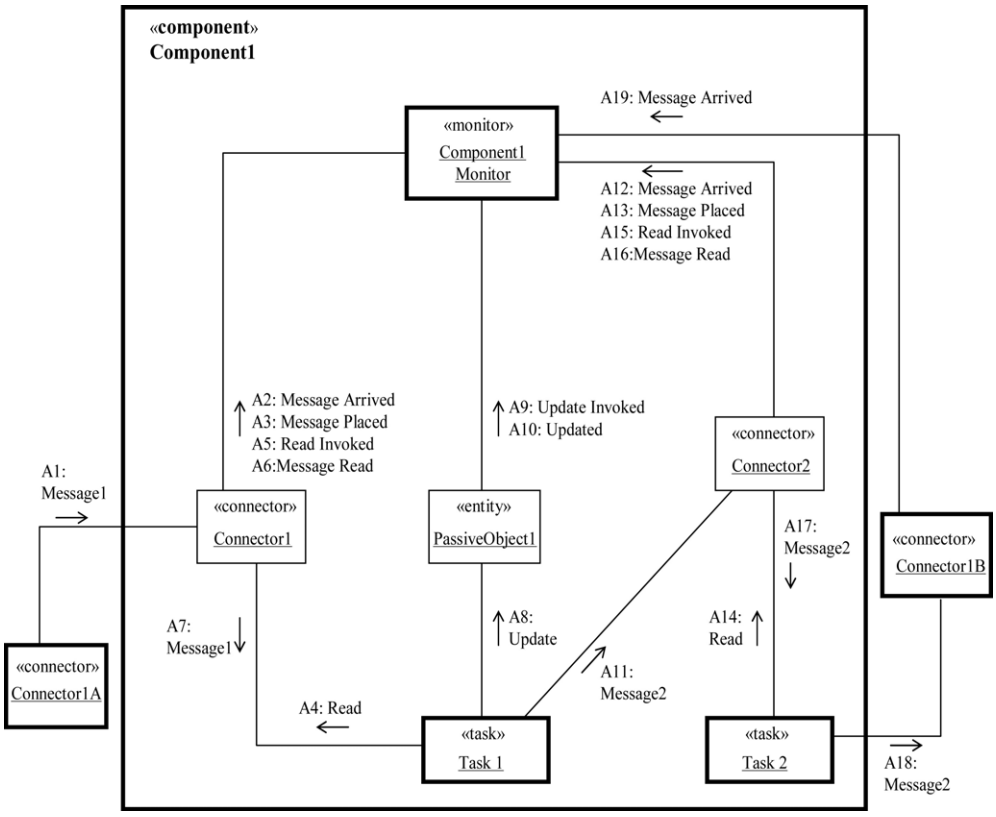

Fig. 3. Component Monitor detecting anomalies in component.

Task1 has called an operation of Connector1 to read a message and is normally working up to that point. If the message, *Read Invoked* (A5 in Fig. 3), is not notified by Connector1 within a specified time interval, the Component1 Monitor begins to suspect an anomaly in Task1. With the notification message, *Message Read* (A6 in Fig. 3), the statechart ((b) in Fig. 4) executes from the *Reading Message* state to the *Processing Message* state. This transition indicates that the "read" operation in Connector1, called by Task1, has executed successfully. Otherwise, the Component1 Monitor may suspect an anomaly in the operation in Connector1. The entity object, *Passive Object1* (Fig. 3), also notifies the Component1 Monitor both when Task1 invokes an operation, *update*, in Passive Object1 to update the entity object (A9 in Fig. 3) and when the operation invoked by Task1 finishes updating the entity object successfully (A10 in Fig. 3). The former notification message ensures that Task1 has processed a message (i.e., the Processing Message state of (b) in Fig. 4) while the latter makes sure that the invoked operation in PassiveObject1 has updated the entity object successfully without any anomaly (i.e., the Updating Passive Object state of (b) in Fig. 4). Similar to Connector1, Connector2 notifies the Component1 Monitor both when a message has arrived from Task1 and when the message is placed in the buffer or queue.

Connectors between components are also involved in detecting anomalies in task threads in components. This will be described in the next section.

5. Anomaly detection among components

Anomalies in message communication between components are detected by monitoring the behavior of connectors between components and networks between connectors (i.e., the communication channel between connectors). Unlike a connector between tasks in a component, connectors between components are active objects, having their own threads of controls, which can initiate actions sending or receiving messages on behalf of components. A network between a sender connector and a receiver connector is also considered to be an active object, which delivers messages between the connectors.

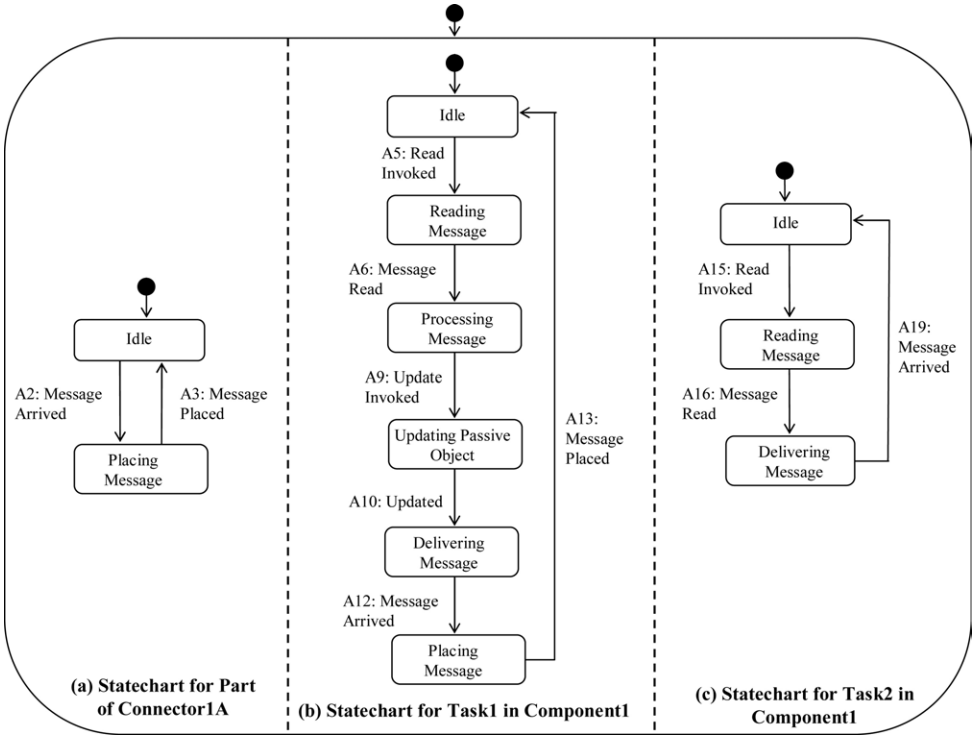

Fig. 4. Statecharts encapsulated in monitor in a component.

The behavior of connectors between components and networks between connectors is traced by the System Monitor using the messages notified by connectors between components. In message communication between two components, a sender connector between components packs a message and sends it to a receiver connector between components, which unpacks the message received and sends it to a connector within a receiver component. A sender connector between components notifies the System Monitor just after it has sent a message to a receiver connector on behalf of a component. A receiver connector also notifies the System Monitor when it receives a message on behalf of a component. The System Monitor encapsulates statecharts for the threads of active objects, that is, connectors between components and networks between connectors, monitoring the behavior of the threads to detect anomalies in message communication between components. It is expected that the statecharts in the System Monitor are executed within bounded time intervals by notification messages from connectors between components.

A connector between components can be classified into an asynchronous connector, a synchronous connector, or connectors encapsulating protocols specific to application systems. An asynchronous connector is used for asynchronous message communication, in which a sender component sends a message and continues without waiting for a response from a receiver component. A synchronous connector is involved in synchronous message communication in which a sender component sends a message to a receiver component and then waits for a reply or acknowledgement. When a receiver component replies to or sends an acknowledgement to a sender component, the sender component continues to work. A receiver component is suspended if no message is available. A connector can encapsulate a complex communication protocol dependent on application systems [6].

An asynchronous (sender) connector, Connector1B (Figs. 1 and 3), and an asynchronous (receiver) connector, Connector3A (Fig. 1), are depicted in Figs. 5 and 6 respectively, which detail the structure of the connector. An asynchronous (sender) connector is structured with Outgoing Message Queue and Call Stub. The Outgoing Message Queue synchronizes the message communication between a task in a component and Call Stub in the connector, notifying the Component Monitor in a component when a message delivery is requested by a task in a component. For example, when Task2 (Fig. 5) in Component1 requests Outgoing Message Queue to deliver a message (B1 in Fig. 5 and A18 in Fig. 3), the Outgoing Message Queue notifies the message arrival (B2 in Fig. 5 and A19 in Fig. 3) to the Component1 Monitor in Component1. The notification message is used to trace the behavior of Task2 in Component1. In the meanwhile, the Outgoing Message Queue notifies the System Monitor of the message arrival from Task2 (B2a

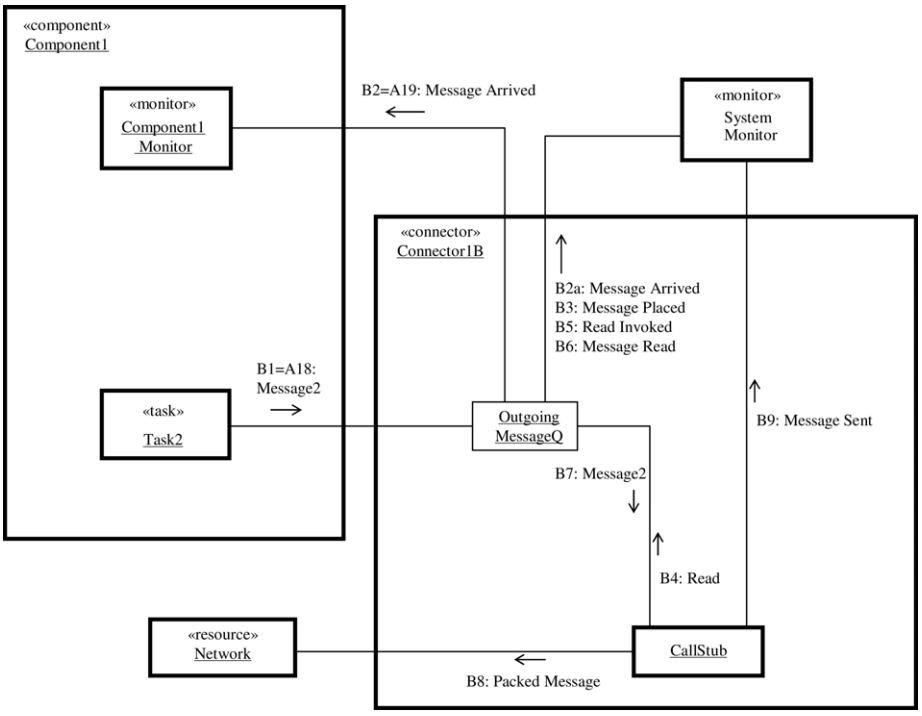

Fig. 5. Asynchronous (sender) connector and message notification.

in Fig. 5) and the status of handling the message in the queue (B3 in Fig. 5), so that the System Monitor detects anomalies in the Outgoing Message Queue.

On the other hand, the Outgoing Message Queue notifies the System Monitor when Call Stub reads a message from the queue (B5 in Fig. 5) and when the read operation has been executed successfully (B6 in Fig. 5). The Call Stub packs a message and sends the packed message to the receiver component via the network. The Call Stub also notifies the System Monitor (B9 in Fig. 5) after it has sent a packed message to the receiver (B8 in Fig. 5).

Fig. 6 depicts an asynchronous (receiver) connector, Connector3A (Fig. 1), which receives a message from Component1 on behalf of Component3. The Incoming Message Queue synchronizes the message communication between the network and Return Stub in the connector, notifying the System Monitor both when any operation in the queue is called by the network or Return Stub (C2 and C5 in Fig. 6) and when the called operation has been executed successfully (C3 and C6 in Fig. 6). The Return Stub unpacks the message delivered through network and sends it to a connector, Connector3, in a receiver component (C8 in Fig. 6), and notifies the System Monitor that a message is sent to the Connector3 (C9 in Fig. 6).

The System Monitor can immediately detect an anomalous object in a connector between components or anomalous network between connectors if an expected notification message does not arrive from connectors between components. Fig. 7 depicts statecharts encapsulated in the System Monitor, which are used to monitor the behavior of Task2 ((a) in Fig. 7), Call Stub in Connector1B ((b) in Fig. 7), the network between Connector1B and Connector3A ((c) in Fig. 7), and the Return Stub in Connector3A ((d) in Fig. 7). When there is no notification message from connectors, its associated statechart describing the behavior of a connector or network does not make a transition within a reasonable time interval. For example, the Call Stub in Connector1B (Fig. 5) is anomalous if the message "B5: Read Invoked" ((b) in Fig. 7) does not arrive within a specified time interval, while the network between Connector1B and Connector3A is anomalous if the message "C2: Message Arrived" ((c) in Fig. 7), does not arrive within a time interval after the message "B9: Message Sent" ((c) in Fig. 7) has arrived at the System Monitor.

6. Evaluation of the approach

The approach to anomaly detection in this paper has been evaluated to validate:

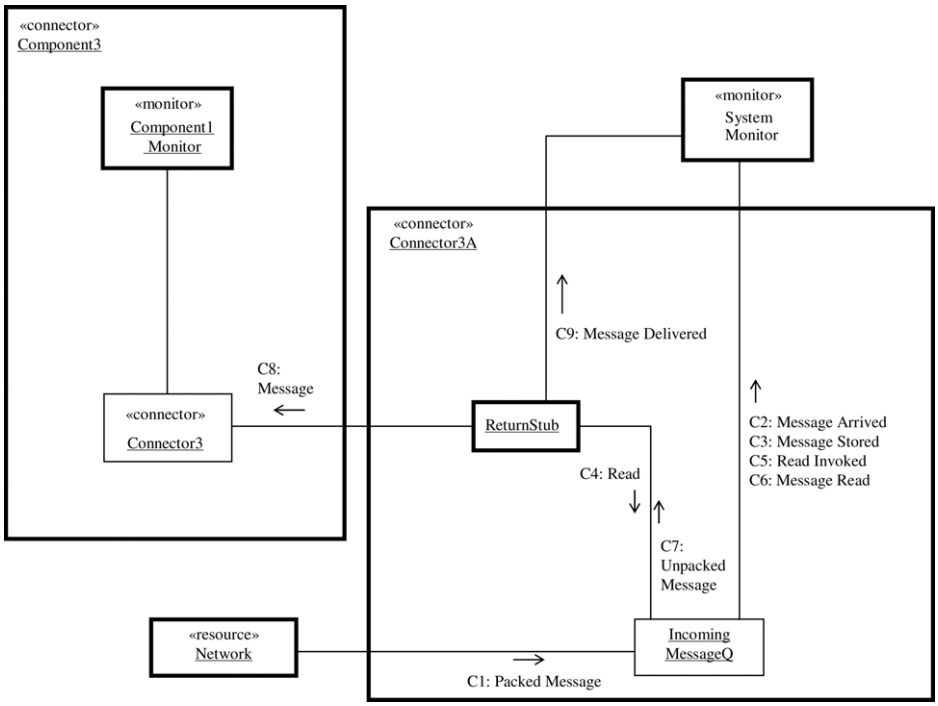

Fig. 6. Asynchronous (receiver) connector and message notification.

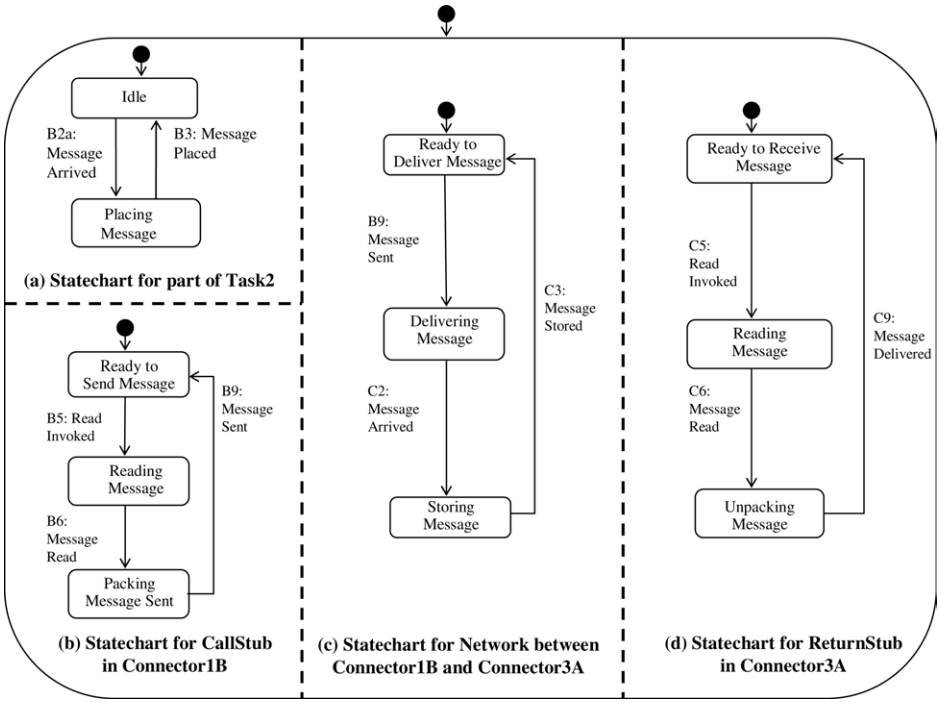

Fig. 7. Statecharts encapsulated in system monitor.

(a) Modeling of Component Monitors and System Monitor. The component monitors within components and System Monitor between components were modeled in a case study using the elevator system to check whether the modeling approach could be used in application systems. The elevator system is structured with Elevator, Scheduler, and Floor components. Fig. 8 depicts part of the Elevator component and its asynchronous (sender)

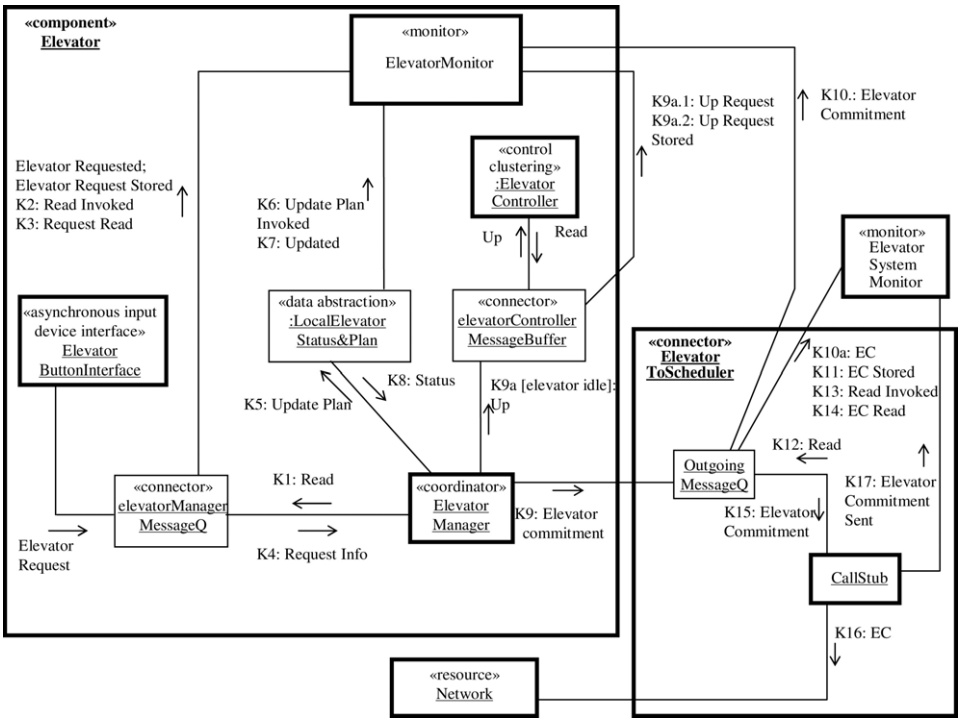

Fig. 8. Elevator component and its asynchronous (sender) connector.

connector to communicate with the Scheduler component. The Elevator Monitor encapsulates statecharts for the task threads in the Elevator Component, whereas the Elevator System Monitor contains statecharts for message communications among Elevator, Scheduler, and Floor components. A message sequence K1 through K10 is associated with the behaviour of the thread of the Elevator Manager task, which is described in statecharts encapsulated in the Elevator Monitor. The notification messages K2 and K3 from the Elevator Manager Message Queue connector, K6 and K7 from the Local Elevator Status & Plan passive object, K9a.1 and K9a.2 from the Elevator Controller Message Buffer connector in the Elevator component, and K10 from the Outgoing Message Queue in the asynchronous *ElevatorToScheduler* connector are used by the Elevator Monitor to monitor the thread of the Elevator Manager task. The notification messages, K10a and K11, are used to determine if an operation in the Outgoing Message Queue, called by the Elevator Manager, has been executed successfully. The messages K13 and K14 from the Outgoing Message Queue and K17 from the Call Stub in the asynchronous *ElevatorToScheduler* connector are notified to the Elevator System Monitor to monitor the Call Stub task in the connector and the network between the Elevator and Scheduler components.

Fig. 9 depicts the Scheduler component and its asynchronous receiver connector. When a message, Elevator Commitment (B1 in Fig. 9), is delivered to the *SchedulerToElevator* connector, the Incoming Message Queue notifies the Elevator System Monitor (B2 in Fig. 9). The message "Elevator Commitment Stored" (B3 in Fig. 9) confirms that the message that has just arrived has been stored successfully in the Incoming Message Queue. The messages B5 and B9 are used by the Elevator System Monitor to check the status of the thread of the Return Stub task, while the message B6 is used to check the status of an operation in the Incoming Message Queue, called by the Return Stub task.

(b) Notification mechanism within components. The notification mechanism within components, which provides a basis for anomaly detection of component monitors, has been evaluated to check if the mechanism can be implemented in both passive objects accessed by tasks and connectors between tasks. For testing this, we implemented two tasks such as a producer task and a consumer task, two connectors such as a message buffer connector (buffer size is 1) and a message queue connector (queue size is n), and an entity (passive) object, in which the producer task sent messages to the message buffer connector and the consumer task received messages from the connector. When an operation of the message buffer connector between the producer and consumer tasks

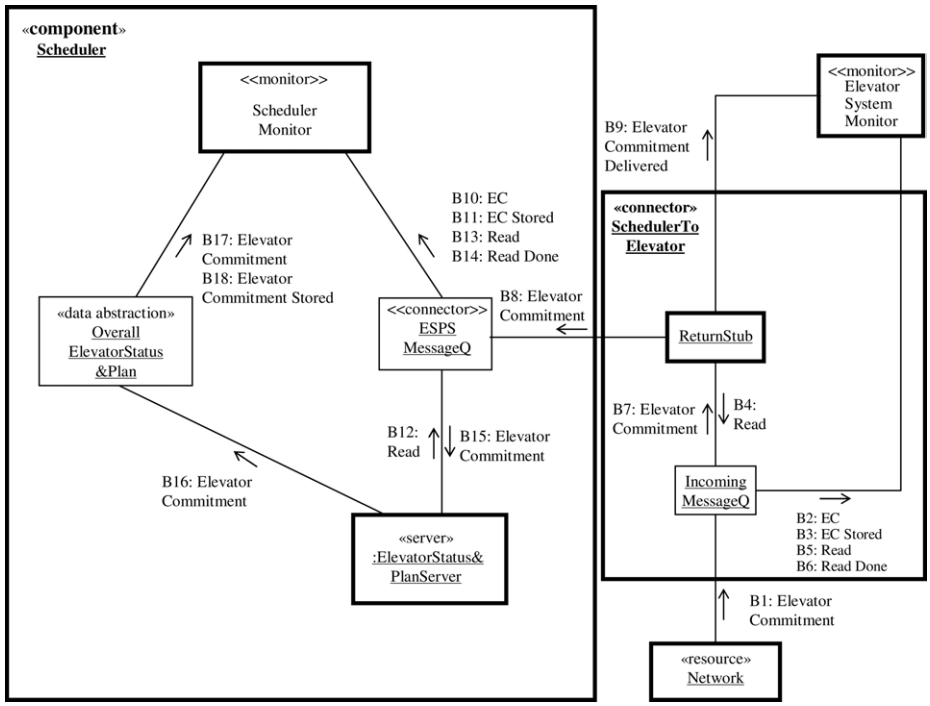

Fig. 9. Scheduler component and its asynchronous receiver connector.

was called by tasks to send or receive a message, the connector sent a notification message to a message queue from which the monitor of a component read the notification messages for detecting anomalies in the producer task, consumer task, and the message buffer connector between the producer and consumer tasks. The entity object also sent notification messages to a message queue when the producer and consumer tasks invoked operations of the object to read or store data. The notification mechanism worked as described in Section 4.

(c) Notification mechanism in connectors between components. The notification mechanism encapsulated in connectors between components has been evaluated to check if the mechanism can be implemented in objects in the connectors. The asynchronous sender connector described in Section 5 was implemented by means of both a message queue (i.e., the Outgoing Message Queue), which is a passive object, and a task (i.e., CallStub), which is a concurrent object. Similarly, the asynchronous receiver connector was made using a message queue (i.e., the Incoming Message Queue) and a task (i.e., the Return Stub). The messages notified from the connectors have also been tested to check if the messages are the same as what the System Monitor expects to detect anomalies in communication between components. The notification mechanism worked as described in Section 5.

7. Related work

Considerable research on fault detection has been peformed in the area of fault tolerant systems using time testing, referred to as “heartbeating” [3,8–10]. This technique depends on message handshaking between a monitor and a component (or subsystem) being monitored, which should be performed at predefined, periodic times. A fault is detected if a heartbeat does not occur within a specified time. This technique can verify that a component still provides functional services to others, but it does not indicate where a fault occurred in a component. Compared with time testing, our approach uses the specifications of components and connectors between components, as well as timing constraints for each notification message, pinpointing where an anomaly occurred in a system.

In [7], faults in a fault-tolerant component are detected using exceptions, which may be generated by the component when it cannot successfully process a service request from others. To achieve this, the exceptions and their conditions for operations of each fault-tolerant component should be defined at design-time. However, most of the software faults remaining after development are unanticipated, state-dependent faults in which a large number of the faults are state-

dependent faults activated by particular input sequences [14]. Our work detects unanticipated, state-dependent faults using statecharts describing the threads of each active object in a system.

An anomaly in a system does not always mean a fault that can lead to a system's failure. In [4], properties of a system, such as the performance of a system, are detected using a set of probes that are deployed in a target system or physical environment. The low-level observations, probes, are converted into system properties of software architectural models. Our current approach cannot detect the change in performance of a system resulting from network congestion and overload in service requests to components, but it may detect those system properties by modifying the timing constraints in the decision mechanism in the component monitor of each component and the system monitor for the network. To achieve this, the decision mechanism may need to have multiple levels of timing constraints on expected notification messages.

8. Conclusions

This paper has described an approach to anomaly detection in software architecture with connectors where anomalies are detected intra-component and inter-component of the architecture. Notification messages from connectors between tasks and passive objects accessed by tasks in a component are used by the Component Monitor within a component to detect anomalies of objects in the component. The System Monitor detects anomalies in message communication between components on the basis of messages notified by connectors between components. The Component Monitor and the System Monitor detect anomalous objects immediately when objects in a system do not behave as specified, pinpointing where the anomalies are in a system—tasks, connectors between tasks, passive objects, connectors between components, and networks between connectors.

The approach to anomaly detection in this paper retains several research issues. The anomaly detection approach suggested in this paper can be extended to increase the accuracy of anomaly detection. The basic ideas for improving the accuracy of detection are to use multiple notification messages and relationships between threads of active objects such as tasks, connectors between components, and networks between connectors. In this paper, we have assumed that a system has a single System Monitor, which may be a bottleneck when the monitor is out of service. Instead of having a centralized System Monitor, an approach to decentralizing multiple System Monitors needs to be studied further.

References

[1] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, Reading, MA, 1999.
[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, Pattern Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.
[3] P. Felber, X. D'efago, R. Guerraoui, P. Oser, Failure detectors as first class objects, in: Proceedings of the International Symposium on Distributed Objects and Applications, DOA'99, Edinburgh, Scotland, 1999, pp. 132–141.
[4] D. Garlan, S.-W. Cheng, B. Schmerl, Increasing system dependability through architecture-based self-repair, in: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems, Springer-Verlag, 2003.
[5] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
[6] H. Gomaa, D.A. Menasce, M.E. Shin, Reusable component patterns for distributed software architectures, in: SSR'01 Sponsored by ACM/SIGSOFT, 18–20 May, Toronto, Ontario, Canada, 2001.
[7] P.A. de C. Guerra, R. de Lemos, An idealized fault-tolerant architectural component, Workshop on Architecting Dependable Systems of ICSE'02, 25 May, Orlando, FL, 2002.
[8] HA Forum, Providing Open Architecture High Availability Solutions, Revision 1.0, February, 2001.
[9] K.H. Kim, C. Subbaraman, The PSTR/SNS scheme for real-time fault tolerance via active object replication and network surveillance, IEEE Trans. Knowledge Data Eng. 12 (2) (2000) 145–159.
[10] K. Mills, S. Rose, S. Quirolgico, M. Britton, C. Tan, An autonomic failure-detection algorithm, in: Proceedings of the 4th International Workshop on Software Performance, WOSP 2004, 14–16 January, San Francisco, CA, 2004.
[11] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language Reference Manual, Addison Wesley, Reading, MA, 1999.
[12] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[13] M.E. Shin, Self-healing component in robust software architecture for concurrent and distributed systems, Sci. Comput. Programming 57 (1) (2005) 27–44.
[14] Wilfredo Torres-Pomales, Software Fault Tolerance: A Tutorial, NASA/TM-2000-210616, October 2000.